**Payment Service**

The **Payment Service** will simulate payment processing and integrate with the **Order Service** to ensure that orders are created only after the payment is successfully captured. Below are detailed step-by-step instructions for implementing this functionality, including the design of the Payment Service, its integration with the Order Service, and the required flow to maintain consistency.

**Step 1: Design the Payment Entity**

1. **Purpose**:
   * The Payment entity represents a single payment attempt or transaction.
   * It will store details about the payment, the customer, the payment reference, and its status.
2. **Fields to Include**:
   * **key**:
     + A user-defined unique identifier for the payment.
   * **customer**:
     + A reference to the customer associated with the payment.
     + Use the customerId or the reference of the User entity.
   * **paymentReference**:
     + A unique reference used by the payment service provider (PSP).
   * **amount**:
     + The total amount the payment intends to receive, typically matching the cart or order gross total.
   * **paymentMethodInfo**:
     + Details about the payment method, including:
       - **Type** (e.g., CreditCard, UPI).
       - **PSP Name** (e.g., Stripe, PayPal).
       - **Date** of the payment attempt.
   * **paymentStatus**:
     + The status of the payment (e.g., SUCCESS, FAILED).
3. **Design Considerations**:
   * Ensure key and paymentReference are unique.
   * Use enums to define the payment method type and status.

**Step 2: Create the Payment Repository**

1. **Purpose**:
   * The repository handles database interactions for the Payment entity.
2. **Required Methods**:
   * Save a new payment.
   * Fetch a payment by its key.
   * Fetch a payment by its paymentReference.
3. **Design Considerations**:
   * Ensure database-level constraints for unique fields (key, paymentReference).

**Step 3: Define DTOs**

1. **PaymentRequest**:
   * Captures the necessary details for initiating a payment.
   * Fields:
     + key, customerId, paymentReference, amount, and paymentMethodInfo.
2. **PaymentResponse**:
   * Structures the response after a payment attempt.
   * Fields:
     + key, paymentReference, amount, paymentMethodInfo, and paymentStatus.
3. **PaymentStatusUpdateRequest**:
   * Captures updates to the payment status.
   * Fields:
     + paymentReference, paymentStatus.

**Step 4: Implement the Payment Processing Logic**

1. **Responsibilities**:
   * Simulate the payment processing logic.
   * Update the payment status (SUCCESS or FAILED) based on the outcome.
   * Notify the **Order Service** of the payment status.
2. **Steps**:
   * **Step 1: Validate Payment Details**:
     + Ensure the key and paymentReference are unique.
     + Validate the amount against the cart or order total.
   * **Step 2: Process Payment**:
     + Simulate payment success or failure using randomization or fixed logic.
     + Update the paymentStatus accordingly.
   * **Step 3: Notify Order Service**:
     + If the payment is successful, send a notification to the Order Service to create the order.
     + If the payment fails, log the failure and return an appropriate response.

**Step 5: Build the REST API for the Payment Service**

1. **Endpoints**:
   * **POST /payments**:
     + Initiates a payment with the provided details.
   * **PUT /payments/{paymentReference}/status**:
     + Updates the payment status.
   * **GET /payments/{key}**:
     + Retrieves the details of a payment using its key.
2. **Implementation Notes**:
   * Use standard HTTP status codes for responses.
   * Validate input data before processing.

**Step 6: Integrate Payment Service with Order Service**

1. **Workflow**:
   * **Step 1**: Initiate Payment
     + When a customer initiates an order, call the Payment Service to create a payment.
     + The Payment Service processes the payment and updates the payment status.
   * **Step 2**: Notify Order Service
     + If the payment is successful:
       - Notify the Order Service to create the order.
       - Pass the payment reference and details to the Order Service for order tracking.
     + If the payment fails:
       - Notify the customer of the failure.
2. **Communication**:
   * Use REST or event-driven messaging (e.g., RabbitMQ, Kafka) for communication between services.
3. **Order Service Logic**:
   * Only create orders for payments with SUCCESS status.
   * Reject order creation requests for failed payments.

**Step 7: Secure the Payment Service**

1. **JWT-Based Authentication**:
   * Validate the JWT token from the API Gateway.
   * Ensure only authorized customers can initiate payments for their own cart or order.
2. **Role-Based Access Control**:
   * Allow only CUSTOMER role to initiate payments.
   * Allow only MANAGER or ADMIN roles to view all payment records (e.g., for auditing).

**Step 8: Testing**

1. **Unit Testing**:
   * Test payment creation with valid and invalid inputs.
   * Test the random or fixed logic for simulating payment success or failure.
2. **Integration Testing**:
   * Test the end-to-end flow from order creation to payment processing and order confirmation.
   * Validate communication between Payment Service and Order Service.
3. **Edge Case Testing**:
   * Test with invalid payment references or keys.
   * Simulate payment failures and ensure proper error handling.

**Sample Payment Workflow**

1. **Customer Perspective**:
   * Customer initiates a payment for their cart or order.
   * If the payment is successful, the order is created, and the customer receives confirmation.
   * If the payment fails, the customer is notified, and they can retry the payment.
2. **Backend Flow**:
   * Payment Service validates and processes the payment.
   * Payment Service notifies the Order Service upon successful payment.
   * Order Service creates the order only after receiving confirmation of successful payment.